EPJ Web of Conferences **245**, 02017 (2020) https://doi.org/10.1051/epjconf/202024502017
*CHEP 2019*

# ROOT I/O compression improvements for HEP analysis

*Oksana* Shadura[1,*] *Brian Paul* Bockelman[2,**] *Philippe* Canal[3,***] *Danilo* Piparo[4,****] and *Zhe* Zhang[1,†]

[1]University of Nebraska-Lincoln, 1400 R St, Lincoln, NE 68588, United States
[2]Morgridge Institute for Research, 330 N Orchard St, Madison, WI 53715, United States
[3]Fermilab, Kirk Road and Pine St, Batavia, IL 60510, United States
[4]CERN, Meyrin 1211, Geneve, Switzerland

**Abstract.** We overview recent changes in the ROOT I/O system, enhancing it by improving its performance and interaction with other data analysis ecosystems. Both the newly introduced compression algorithms, the much faster bulk I/O data path, and a few additional techniques have the potential to significantly improve experiment's software performance.
The need for efficient lossless data compression has grown significantly as the amount of HEP data collected, transmitted, and stored has dramatically increased over the last couple of years. While compression reduces storage space and, potentially, I/O bandwidth usage, it should not be applied blindly, because there are significant trade-offs between the increased CPU cost for reading and writing files and the reduces storage space.

## 1 Introduction

In the past years, Large Hadron Collider (LHC) experiments are managing about an exabyte of storage for analysis purposes, approximately half of which is stored on tape storages for archival purposes, and half is used for traditional disk storage. Meanwhile for High Luminosity Large Hadron Collider (HL-LHC) storage requirements per year are expected to be increased by a factor of 10 [1].

Looking at these predictions, we would like to state that storage will remain one of the major cost drivers and, at the same time, the bottlenecks for HEP computing.

The new storage and data management techniques, as well as a compression algorithms, are likely will be more required to remove a storage and analysis computing cost bottleneck. It will allow to handle expected data ratios and data volumes needed to be processed by experiments during HL-LHC[1].

Looking into innovative compression algorithms could help to resolve some problems, such as speeding up the user analysis, improving decompression speed, while maintaining the same or better compression ratio. Zstandard [5] is a dictionary-type algorithm (LZ77) with a large search window and fast implementations of entropy coding stage, using either fast

*e-mail: oksana.shadura@cern.ch
**e-mail: bbockelman@morgridge.com
***e-mail: pcanal@fnal.gov
****e-mail: dpiparo@cern.ch
†e-mail: zhan0915@huskers.unl.edu

Finite State Entropy (tANS) or Huffman coding. Zstandard, referred to as ZSTD, is a much more modern compressor then ZLIB, which was initially implemented in 1995, and offers higher compression rates while using less CPU compared to other compression algorithms (e.g. LZMA). ZSTD is available as a ROOT supported compression algorithm, starting from ROOT v6.20 release [3].

## 2 Background

Three years ago, Facebook [7] open-sourced ZSTD, widely used in its software projects. It is largely supported by the community and enhanced by ZSTD authors, who released a variety of advanced capabilities, such as improved decompression speed and better compression ratios.

The initial promise of ZSTD was that it allows users to replace their existing data compression implementation, such as ZLIB, for one with significant improvements on compression speed, compression ratio, and decompression speed. [6]

In addition to replacing ZLIB, ZSTD has taken over many of the tasks that traditionally relied on fast compression alternatives. Fastest compression is still provided by fastest compression settings of LZ4, while ZSTD provides a twice size better compression ratio and still. According to reports from the community, it is slowly replacing the strong compression scenarios previously served by XZ (LZMA) [2], with the benefit of 10 times faster decompression speed.

One of the advanced features of Zstandard is a training mode. It can use a "dictionary" format to make compression of files of an already known type in a more efficient way, for example since baskets within a branch hold similar data, using a common dictionary could be very efficient. If a dictionary is "trained" on an example set of email messages, anyone with access to the dictionary will be able to more efficiently compress another email file. The trick is that the commonalities are kept in the dictionary file, and, therefore, anyone wishing to decompress the email must have already had that same dictionary sent to them [2]. A dictionary is a file that stores the compression settings for small files. Compression dictionary is assembled from a group of typically small files that contain similar information, preferably over 100 files. For the best efficiency, their combined size should be about one hundred times the size of the dictionary produced from them. In general, the smaller the file, the greater the improvement in compression. According to the ZSTD manual page, a dictionary can only increase the compression of a 64KB file by 10 percent, compared with a 500 percent improvement for a file of less than 1KB [6]. The creation of the dictionary can follow two different approaches: dictionary training or dictionary re-utilization. Training of dictionary in ROOT is one of the future work items and will be not discussed in this paper.

## 3 Evaluation of simple ZSTD algorithm for LHC datatsets

In this section, we will try to focus on the evaluation of compression of most used analysis-related formats in CMS, MiniAOD [8] and NanoAOD [9], as well as a simple case of analysis file used by the LHCb experiment.

The MiniAOD is a high-level CMS data file that was introduced in 2014 to serve the needs of the mainstream physics analyses while keeping a small event size - only 30-50 KB per event. It is not readable with bare ROOT (without custom I/O streamers and dictionary support) and requires a special CMSSW setup to be able to read it. Meanwhile, NanoAOD format consists of a flat, ROOT Ntuple-like format, readable with bare ROOT and containing the per-event information that is needed in most generic analyses. It is mostly populated by

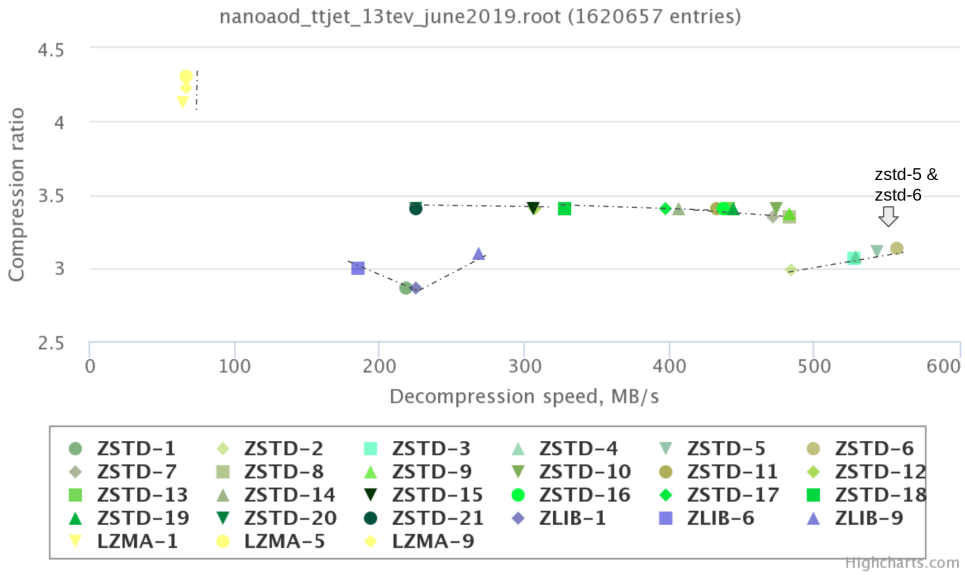


**Figure 1.** Comparison of compression ratio and decompression speed for ZLIB, LZMA and ZSTD algorithms for NanoAOD 2019 file

the basic types floats, double or int and its size per event is the order of 1KB. NanoAODs are usually centrally produced or even produced on-demand with different variations of features or columns required by different physics analysis groups. Users can as well easily extend NanoAOD for their specific studies making a private production when needed. For CMS NanoAOD files, using ZSTD could be a better compromise between size of file on a disk and decompression speed for a faster analysis as well as better compression ratio and $2x$ faster decompression than ZLIB and $6x$ faster compared to LZMA, while file compressed with ZSTD is only 20 % bigger size (all results are shown on the Figure 1 and 2). 20 % bigger size could be a significant difference for RAW data format, since these files are not accessed that often, but for analysis-targeted NanoAOD files, the main priority is to enable a faster analysis, improving speed for reading events.

For MiniAOD, measured time spend in decompressing on readback is $15x$ less compared to LZMA, while the size of the file with ZSTD is only 10% bigger.

In case of LHCb, for the very simple NTuples with a simple structure, the best choice could be LZ4 compression algorithm, offering $10x$ time faster read speed (all results are shown on the Figure 3).

In ROOT, the serialization of variable-sized data (for example the C-style arrays) produces two internal arrays: one array contains the branch populated by data of each of the events while the other contains the offset of bytes (memory layout) for each of the events in this branch. LZ4 compression algorithm achieves its performance by looking for byte-aligned patterns, as opposed to ZLIB compression algorithm, which works on individual bits and lacks the Huffman encoding pass, this results in the offset array sequence being effectively incompressible using LZ4. ZSTD has no problems with compression of data that contains the byte offset of each event in the branch data (all results are shown on the Figure 4).

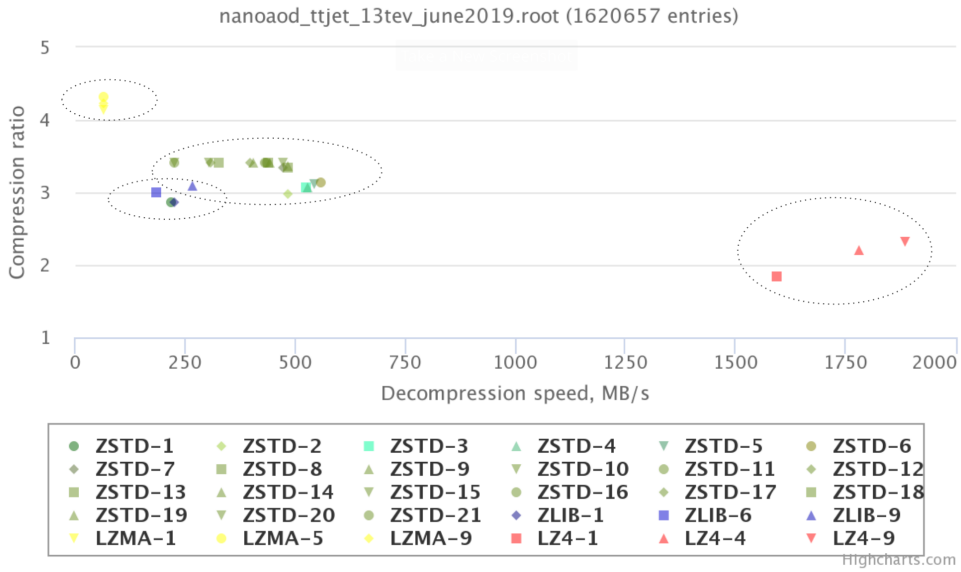


**Figure 2.** Comparison of compression ratio and decompression speed for all compression algorithms for NanoAOD 2019 file.

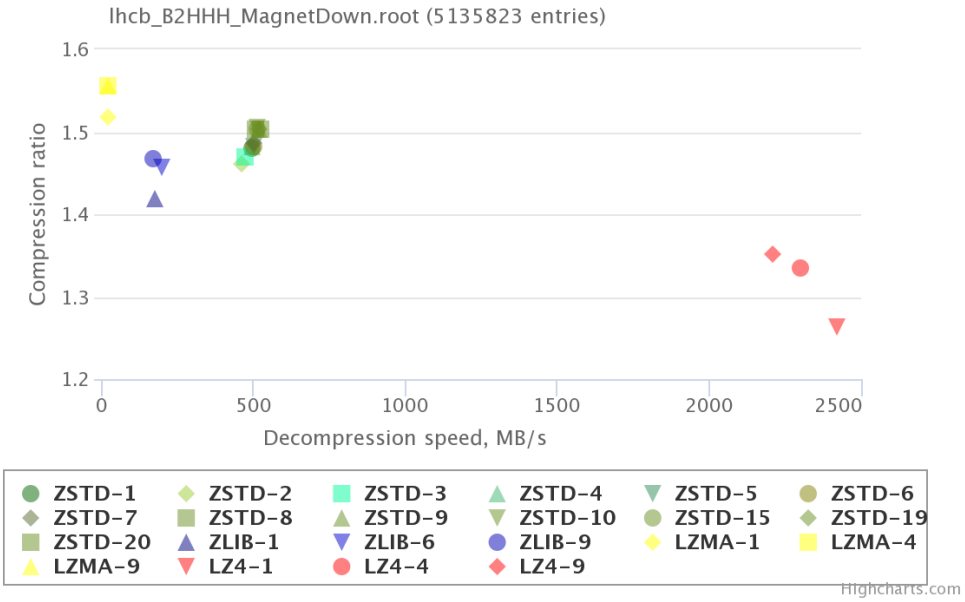


**Figure 3.** Comparison of compression ratio and decompression speed for all compression algorithms for LHCb file.

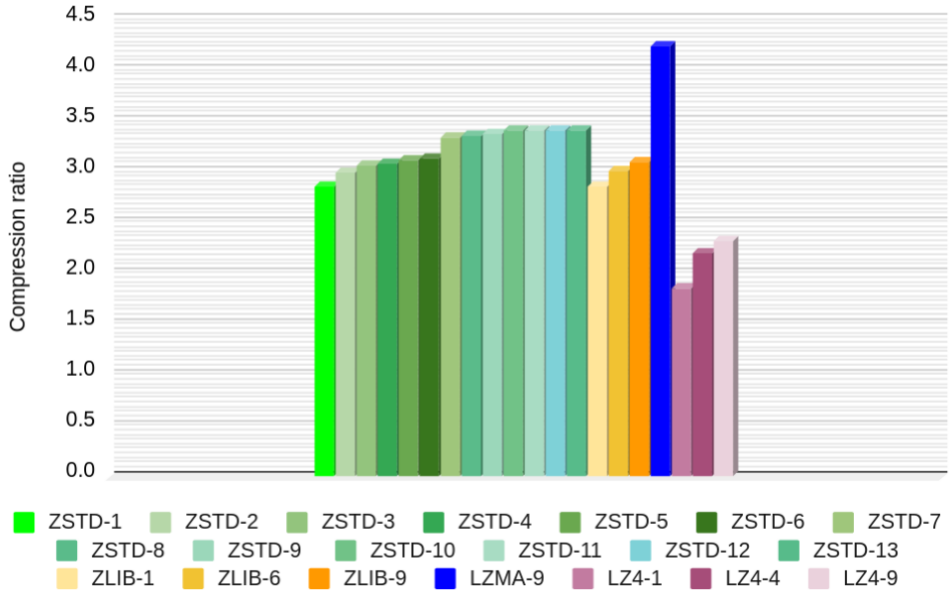


**Figure 4.** Comparison ratio comparison for custom analysis file with variable-sized data (containing C-style arrays).

## 4 `TTree::kOnlyFlushAtCluster` option, offering faster decompression

TTrees can be forced to create only the new baskets at event cluster boundaries, using a `TTree::kOnlyFlushAtCluster` feature. It simplifies file layout and I/O at the cost of memory. For example for the `TTree::kOnlyFlushAtCluster` feature tests shown in Figure 5, NanoAOD 2017 was bigger only by 3.6 % of size, while decompression speed is improved almost up to 200 MB/s [10].

`TTree::kOnlyFlushAtCluster` is recommended for simple file formats such as ntuples where it can show really interesting improvements, but not for more complex data types.

## 5 Limitations and Future work

Some time ago, Bitshuffle pre-conditioner was demonstrated as a possible pre-conditioner for LZ4 implemented in case of lossless compression in ROOT. To improve the performance of LZ4 in this case, we investigated the combination of LZ4 with various “pre-conditioners”. Pre-conditioners transform the sequence of input bytes according to a simple, deterministic algorithm before applying the compression algorithm. [10]

We investigated, inspired by the example of Blosc library [13], a BitShuffle algorithm. This pre-conditioner rearranges the input array’s bytes by reading through the data using fixed strides. The resulting output of the pre-conditioner often contains long sequences of repeated bytes, improving the compression ratio for LZ4. One of the issues exposed was that it is difficult for ROOT to compress its buffers now due to its 9-byte header [10].

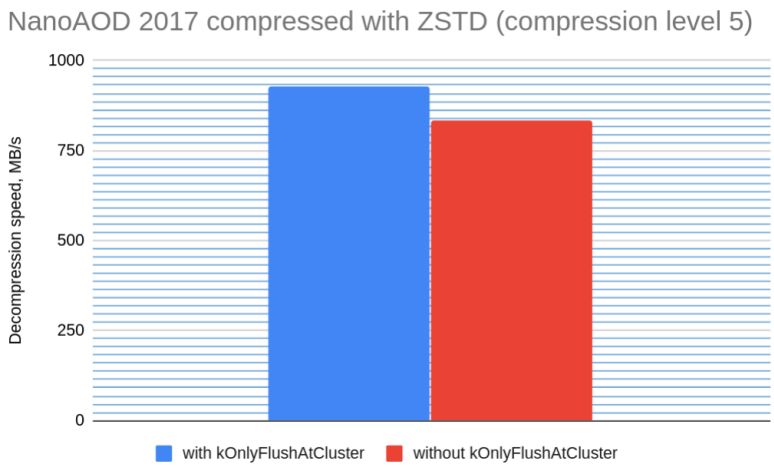


**Figure 5.** Comparison of decompression speed for two file samples NanoAOD 2017, with and without `TTree::kOnlyFlushAtCluster` option.

The idea of using pre-conditioners could be easily expended to be used with other algorithms, such as ZSTD. The next goal of the project will be to validate the possibility to use pre-conditioners in the ROOT compression layer used to compress both ROOT file formats (TTree and RNTuple) for the fastest ROOT compression algorithms: LZ4, ZSTD.

Another interesting investigation could be to extend pre-conditioners to adopt a new BYTE_STREAM_SPLIT [14] encoding from Apache Arrow that improves compression ratio and compression speed for certain types of floating-point data where the upper-most bytes of values do not change much. The existing compressors and encodings in ROOT do not perform well for such data due to noise in the mantissa bytes. The new encoding improves results by extracting the well compressible bytes into separate byte streams which can be afterward compressed by a compressor like ZSTD [15].

## 6 Conclusions

ZSTD has been successfully evaluated and it is ready to be used for compression of data analysis formats by anyone who has interest in it. We would like to encourage the LHC experiments to try ZSTD compression algorithm, which is already available in ROOT and share their feedback about it.

## 7 Acknowledgments

This work has been supported by U.S. National Science Foundation grants OAC-1450323.

## References

[1] Elsen, Eckhard. "A Roadmap for HEP Software and Computing R&D for the 2020s." (2019): 16.

[2] XZ Utils. https://tukaani.org/xz/. Accessed 6 Mar. 2020.

[3] R. Brun, F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Nucl. Inst. & Meth. in Phys. Res. A **389** (Proceedings AIHENP'96 Workshop,1997).

[4] Facebook Github organization. GitHub, https://github.com/facebook. Accessed 22 Feb. 2020.

[5] Facebook/Zstd. 2015. Facebook, 2020. GitHub, https://github.com/facebook/zstd.

[6] Collet, Y., and M. Kucherawy. "Zstandard Compression and the application/zstd Media Type." RFC 8478 (2018).

[7] "Zstandard: How Facebook Increased Compression Speed." Facebook Engineering, 19 Dec. 2018, https://engineering.fb.com/core-data/zstandard/.

[8] Petrucciani, Giovanni, Andrea Rizzi, and Carl Vuosalo. "Mini-AOD: A new analysis data format for CMS." Journal of Physics: Conference Series. Vol. 664. No. 7. IOP Publishing, 2015.

[9] Rizzi, Andrea, Giovanni Petrucciani, and Marco Peruzzi. "A further reduction in CMS event data for analysis: the NANOAOD format." EPJ Web of Conferences. Vol. 214. EDP Sciences, 2019.

[10] Shadura, Oksana, and Brian Paul Bockelman. "ROOT I/O compression algorithms and their performance impact within Run 3." arXiv preprint arXiv:1906.04624 (2019).

[11] Canal, Philippe, Brian Bockelman, and René Brun. "ROOT I/O: The fast and furious." Journal of Physics: Conference Series. Vol. 331. No. 4. IOP Publishing, 2011.

[12] "Bitshuffle" Github https://github.com/kiyo-masui/bitshuffle Accessed 15 Jul. 2020.

[13] "Blosc, an extremely fast, multi-threaded, meta-compressor library." https://blosc.org/pages/ Accessed 15 Jul. 2020.

[14] "Apache BYTE_STREAM_SPLIT encoding." GitHub, https://github.com/apache/arrow/blob/master/cpp/src/arrow/util/byte_stream_split.h Accessed 15 Jul. 2020

[15] "Apache/Parquet-Format." GitHub, https://github.com/apache/parquet-format. Accessed 6 Mar. 2020.